Semantics of Programming Languages
Exercise Sheet 6

Exercise 6.1 Compiler optimization

A common programming idiom is \textit{IF} \textit{b} \textit{THEN} \textit{c}, i.e., the else-branch consists of a single \texttt{SKIP} command.

1. Look at how the program \texttt{IF Less (V "x") (N 5) THEN "y" := N 3 ELSE SKIP} is compiled by \texttt{ccomp} and identify a possible compiler optimization.

2. Implement an optimized compiler (by modifying \texttt{ccomp}) which reduces the number of instructions for programs of the form \textit{IF} \textit{b} \textit{THEN} \textit{c}.

3. Extend the proof of \texttt{comp_bigstep} to your modified compiler.

Exercise 6.2 Type coercions

Adding and comparing integers and reals can be allowed by introducing implicit conversions: Adding an integer and a real results in a real value, comparing an integer and a real can be done by first converting the integer into a real. Implicit conversions like this are called \textit{coercions}.

1. Modify, in the theory \textit{HOL–IMP. Types} (copy it first), the inductive definitions of \texttt{taval} and \texttt{tbval} such that implicit coercions are applied where necessary.

2. Adapt all proofs in the theory \textit{HOL–IMP. Types} accordingly.

\textit{Hint}: Isabelle already provides the coercion function \texttt{real_of_int} (\texttt{int} $\Rightarrow$ \texttt{real}).
Homework 6.1  Pairs

Submission until Monday, December 6, 10:00am.

In this exercise, we extend the expression language of IMP with pair values.

```haskell
datatype val = Iv int | Pv val val

type_synonym vname = string

type_synonym state = “vname ⇒ val”

datatype aexp = N int | V vname | Plus aexp aexp | Pair aexp aexp
```

Complete the following inductive predicate for evaluating expressions:

```haskell
inductive taval :: “aexp ⇒ state ⇒ val ⇒ bool”

\("taval\ (N i) s (Iv i)\) | "taval\ (V x) s (s x)\) |

It should also be able to add pairs. In this case, the addition should be performed pair-wise. This should also work for nested pairs.

For simplicity, we do not modify Boolean expressions. Less can only compare two integer values:

```haskell
datatype bexp = Bc bool | Not bexp | And bexp bexp | Less aexp aexp

datatype bexp = Bc bool | Not bexp | And bexp bexp | Less aexp aexp
```

We add an assignment construct for pairs \((x, y) ::= a\) to the command language:

```haskell
datatype com = SKIP |
| Assign vname aexp (”_ := _” [1000, 61] 61) |
| AssignP vname × vname” aexp (”_ := _” [1000, 61] 61) |
| Seq com com (“;” [60, 61] 60) |
| If bexp com com (”IF _ THEN _ ELSE _” [0, 61] 61) |
| While bexp com (“WHILE _ DO _” [0, 61] 61) |
```

Adopt the small-step semantics accordingly:

```haskell
inductive small_step :: “(com × state) ⇒ (com × state) ⇒ bool” (infix “⇒” 55)
```

We now also add a pair type to the typing system:

```haskell
datatype ty = Ity | Pty ty ty
```
type synonym tyenv = "vname ⇒ ty"

Complete the typing rules:

inductive atyping :: "tyenv ⇒ aexp ⇒ ty ⇒ bool" ("\(\Gamma, N : Ity \vdash \_\)"
\(\Gamma, V x : \Gamma x \vdash \_\)"
\(\Gamma, Ic ty : \Gamma : \_\)"
\(\Gamma, Bty : \Gamma : \_\)"
\(\Gamma, And ty : \Gamma : \_\)"
\(\Gamma, Skip ty : \Gamma : \_\)"
\(\Gamma, Assign ty : \Gamma : \_\)"
\(\Gamma, Seq ty : \Gamma : \_\)"
\(\Gamma, If ty : \Gamma : \_\)"
\(\Gamma, While ty : \Gamma : \_\)"

This function determines the type of a value:

fun type :: "val ⇒ ty" where
"type (Iv i) = Ity" |
"type (Pv v1 v2) = Pty (type v1) (type v2)"

lemma type_eq_Ity[simp]: "type v = Ity ⇔ (\(\exists i. v = Iv i\))" by (cases v) simp_all

Hint: You will also need a similar lemma for Pty t1 t2.

definition styping :: "tyenv ⇒ state ⇒ bool" (infix "\(\vdash\)"
where "\(\Gamma \vdash s \equiv (\forall x. \_\)"

Complete the proofs of preservation and progress:

lemma apreservation:
"\(\Gamma \vdash a : \tau \Rightarrow \_\)"
lemma aprogress: "\(\Gamma \vdash a : \tau \Rightarrow \Gamma \vdash s \Rightarrow \_\)"
lemma bprogress: "\(\Gamma \vdash b \Rightarrow \Gamma \vdash s \Rightarrow \_\)"

theorem progress:
"\(\Gamma \vdash c \Rightarrow \Gamma \vdash s \Rightarrow \_\)"

theorem styping_preservation:
"\(\Gamma \vdash c \Rightarrow \Gamma \vdash s \Rightarrow \_\)"

theorem ctyping_preservation:
"\(\Gamma \vdash c \Rightarrow \Gamma \vdash s \Rightarrow \_\)"

abbreviation small_steps :: "com * state ⇒ com * state ⇒ bool" (infix "\(\Rightarrow\)"


where “x →∗ y == star small_step x y”

Finally, we can recover the proof of type-soundness:

**theorem** type_sound:

“(c,s) →∗ (c’,s’) ⇒ Δ ⊢ c ⇒ Δ ⊢ s ⇒ c’ ≠ SKIP

⇒ ∃cs''. (c’,s’) → cs’’”

apply (induction rule: star_induct)

apply (metis progress)

by (metis typing Preservation ctyping preservation)

---

**Homework 6.2  Continue**

Submission until Monday, December 2, 10:00am.

Your task is to add a continue command to the IMP language. The continue command should skip all remaining commands in the innermost while loop.

The new command datatype is:

```plaintext
datatype com = SKIP
  | Assign vname aexp (“::=” [1000, 61] 61)
  | Seq com com (“;;/” [60, 61] 60)
  | If bexp com com (“(IF / THEN / ELSE /)” [0, 0, 61] 61)
  | While bexp com (“(WHILE / DO /)” [0, 61] 61)
  | CONTINUE
```

The idea of the big-step semantics is to return not only a state, but also a continue flag, which indicates that a continue has been triggered. Modify/augment the big-step rules accordingly:

```plaintext
inductive big_step :: “com × state ⇒ bool × state ⇒ bool” (infix “⇒” 55)
```

Your next task is to adopt the compiler such that CONTINUE is also supported. The new compiler will have the following signature:

```plaintext
fun ccomp :: “com ⇒ nat ⇒ instr list” where
```

The extra argument keeps track of the offset from the head of the last preceding while-loop.

To improve automation, first prove the following lemma:

```plaintext
definition “len_of c = length (ccomp c 0)”

lemma length_ccomp[simp]:
  “length (ccomp c i) = len_of c”
```
Now show that your new compiler is correct. To do so, prove the following modified correctness lemma. The modified lemma adds an instruction prefix $pre$, which you can think of as the list of instructions that separates the current instruction and the last loop head.

Note that the original proof made us of heavy automation that is likely going to break after making the changes from above. Use Isar to explore the proof in more detail.

lemma ccomp_bigstep1:
\n\[(c,s) \Rightarrow (f, t) \Rightarrow i \leq length pre\]
\[\Rightarrow pre @ ccomp c i \vdash (length pre, s, stk) \Rightarrow (if f then length pre - i else size(pre @ ccomp c i), t, stk)\]

Finally, re-prove the old correctness theorem:

corollary ccomp_bigstep:
\n\[(c,s) \Rightarrow (False, t) \Rightarrow ccomp c 0 \vdash (0, s, stk) \Rightarrow (size(ccomp c 0), t, stk)\]